URL: http://www.elsevier.nl/locate/entcs/volume65.html 11 pages

Middleware Support for Transparent Client-Side Caching

Christoph Pohl, Alexander Schill [1]

Dresden University of Technology
Institute for System Architecture, Chair for Computer Networks

Abstract

This paper reflects on existing caching concepts in proxies and stubs of component technologies and lines out their advantages and deficiencies. A new concept is introduced that averts proliferation of component stubs on client side while transparently providing efficient caching of attributes that doesn't require any changes to existing code at all. Common object-oriented design facilities and code generation tools are leveraged to support seamless integration of these caching concepts into the development cycle.

1 Introduction

Technologies like remote procedure calls have once been introduced to provide transparent programmatic access to remote interfaces just like to their local equivalents. But reality has proven that transparent distribution is a myth. For building scalable applications it is important to keep in mind that remote calls are far more expensive. This gains even more significance with today's communication intensive component models like Enterprise JavaBeans.

There are different strategies for approaching this problem of network traffic reduction. In the context of component technology, a large portion of remote calls serves the purpose of state transfer where component attributes are queried individually. An interface redesign can help to reduce the granularity of these state transfers. This eliminates several round-trips at the cost of data marshalling which is typically a good trade-off. But as most data is more often read than written, replication is another option to maintain scalability. *Caching* is a special form of partial replication that is often chosen to close the gap between information sources and sinks. It builds up on the assumption that once accessed data will be queried again in the nearer future.

[1] eMail: {pohl,schill}@rn.inf.tu-dresden.de

This differentiates caching from *prefetching* – another variety of partial replication – which tries to load data in advance that might be queried next. The efficiency of replication is mainly determined by the ratio of read and write operations because write operation introduce inconsistencies between replicas of the same object which result in additional need for communication. Further influencing factors include network latency, amount of changed data per write operation, available memory for replication, applications' tolerance to minor inconsistencies etc.

Caching technology can be integrated into component based software in at least two different ways: (1) Using special design patterns, component interfaces can be augmented with necessary methods. This usually implies major redesign for existing applications although it may be a viable option for emerging programs. (2) Orthogonal facilities can be implemented by the component's middleware itself. Existing components remain mostly untouched. Caching requirements are specified in a descriptive manner which can be evaluated by the middleware.

No matter what approach is taken in a particular environment, for component based software there are at least three issues that have to be handled by the caching subsystem: (1) *Component State Caching* comprises providences for local caching of queried component attributes. (2) *Identity of Component Instances* is important to decide whether certain queries refer to the same shared object or component. (3) *Multiple Reference Handling* avoids duplicate cache entries for the same component instance. Components are often related to each other. These relations are typically exposed quite similar to attributes on their interfaces. This implies that there is often more than one way to acquire references to component instances. But all references have to be redirected to the same cached copy within a client process.

A possible way to tackle these problems is introduced in Sect.3. Especially the last point is often ignored by simple distributed caching solutions. Only few authors realize the importance of this issue.

2 Related Work

Caching is not a new feature in object oriented systems. Existing solutions date back to *Orca* [1] and *Shadows* [5] which builds up *Arjuna*, among others.

This article concentrates on the particular issues of component interfaces and proxy objects in an Enterprise JavaBeans setting where hardly any support for caching is available from existing containers.

Many solution providers rely on special client-side proxy objects for optimized bean access. It is important to keep in mind that "client side" refers only to the user of an interface. In the context of EJB, this might as well be a web server processing JSPs and Servlets by querying an EJB container. These two processes don't necessarily run on the same network node. It is rather common to distribute them across the ASP's LAN. Considering the

consequences, the scope of caching proxies is much wider than simple Java Applets and application clients.

Proxy is a common pattern in object-oriented software design [16,8,4]. It refers to objects installed as representatives for some (remote) delegate which they control and whose interface they follow. This approach is typically taken by RPC-style middleware like RMI and IIOP where proxies, a.k.a. *remote stubs*, transparently handle marshalling of arguments and return values, among other tasks.

Distributed applications base on this abstraction level to implement their specific functionality. As anticipated in Sect.1, this leaves us with two possibilities for caching implementations: Either on top this abstraction as an extension to application interfaces or below as additional middleware service.

2.1 Application Level Solutions

Caching implementations based on the abstraction level of component interfaces have to introduce a proxy layer encapsulating caching logic for other client application modules. The closer this proxy layer follows the original component's interfaces, the less modifications to existing client modules are needed. This approach has been followed by *OORPC* [20] and *MinORB* [12], among others. But full transparency is not achievable due to the need for explicit invocation or creation of the proxy objects – client programmers have to be aware of these changes.

The described caching layer can simply try to keep copies of query results but usually several queries are triggered in sequence, e.g. because more than one attribute of a component is needed by client applications to do their work. The caching layer could use *State Object* or related design patterns where a component's state is transfered to the manipulating (client) process in a coarse-grained manner which reduces the overall number of network round-trips. This bulk state transfer can either be triggered by the client application after some modifications or even transparently by the caching layer itself which would add a simple prefetching functionality.

2.1.1 State Object Pattern

The State Object pattern is not to be confused with the behavioral State pattern as introduced in [8]. It is rather a structural extension of the *Proxy* pattern whose main part is formed by a simple holder object that is used for bundled state transfer between an abstract client server pair. This strategy usually saves a number of network round-trips as several attributes are queried together instead of fetching them individually. Other names for the same concept are *Value Object* [14] and *Data Array*.

This pattern uses fixed data structures for state transfer in its original form but extended, more flexible versions like *Dynamic Property* enable clients to query a variable set of the business object's properties instead of the bulk of

attributes. Different use cases become possible at runtime without changes to the component's code.

A similar solution called *Access Beans* is integrated in IBM's Enterprise JavaBeans product line [17]. *Access Beans* is the name of the JavaBeans-style proxies that clients have to use to access EJBs. *Copy Helpers* and *Rowsets* add caching functionality for single and multiple EJB instances in one Access Bean.

Astral Clones [15] use a different approach: Bean classes are made available for use outside of a container. Serialized copies containing the component's state can be transfered to clients by additional methods. As these classes already contain all necessary data manipulation logic, they can be used by clients just like their remote equivalents. Unfortunately, required precautions seem to introduce more problems than they solve and furthermore, the concept won't work with EJB 2.0 entities [7] due to changed persistent state management.

2.1.2 Session Bean Wrapper

Session Facade [14] or *Session Bean Wraps Entity Beans* as in [3] is another pattern that can also be used when it is not possible or desirable, e.g. for compatibility reasons, to modify existing components directly to support methods for the above mentioned Value Object queries. The Session Facade could then parenthesize access to existing components for new clients and add support for these bundled object state queries. Individual component attribute queries would then happen locally on the server running the container of both the entity and the Session Facade component. Client-side proxies could benefit from this added functionality to make their caching more efficient as described above.

2.2 Middleware-based Concepts

As mentioned introductorily, all the above presented concepts require a considerable amount of recoding on client and server side which poses a violation of transparency that might not be feasible in a lot of today's business scenarios. This section introduces alternative ways to accomplish transparent caching.

2.2.1 Distributed Shared Objects

Quite a number of scientific publications in the past decade elaborated on *Distributed Shared Objects*, e.g.*Javanaise* [9]. Not being bound by the constraints of particular middleware or component platforms, most of these proposals managed to build efficient solutions for distributed applications. Unfortunately, hardly any of them managed to prevail or to gain a broader base of acceptance which kept application developers to refrain from investing time and money into these technologies.

2.2.2 Smart Proxies

In the Corba world endeavors for a higher level of transparency have led to the concept of so-called *Smart Proxies* [2] – user defined remote stubs that can be used instead of the ORB's default implementations. These Smart Proxies may contain caching functionality, among others. However, this concept still exists only as provider-specific [3], non-interoperable extensions.

2.2.3 Stub Annotation

Java's RMI and Java-IDL don't have any providences for this concept. Nevertheless, proposals have been made how to fill this gap: *Smart Stubs* [11] is a simple proof-of-concept whose basic idea relies on renaming the system's default remote stubs and replacing them by derived, augmented classes that wrap functionality for caching and performance monitoring.

2.2.4 Caching Services

Caching is a technically motivated aspect that is never directly related to application requirements. It rather emerges out of performance considerations taken at application run-time. Therefore it's only equitable to disburden application programmers of that challenge and to provide caching support that can be configured during deployment without recoding. Other authors had the same feelings which led to projects like *Flex* [10], a CORBA-based framework for client-side caching, and *Cascade* [6], a CORBA service for object caching in WANs.

2.3 Conclusion

It seems that existing solutions have a number of drawbacks: They either force programmers to adapt new programming models or they provide insufficient transparency to client and server programs with the impact that typical caching problems have to be solved by application programmers. Some give little to no regard to global identity of server objects, i.e. multiple references to a single server object of the same client.

3 Middleware Support for Transparent Client-Side Caching

In this section we outline our current work on a transparent client-side caching solution in the context of Enterprise JavaBeans.

3.1 Multiple References to the Same Remote Object

The explosive proliferation of proxies is an often untended problem: Wherever (remote) operations return proxies as remote references, new proxies are being

[2] explained in more detail e.g. in [18]
[3] e.g. Borland / Inprise / Visigenic's VisiBroker and Iona's Orbix provide Smart Proxies

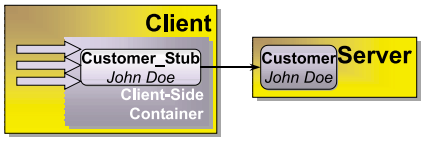

Figure 1. Client-Side Container

created in the client process upon every call. Memory consumption increases linearly with the number of object references. Every client has to check its received proxies for equality and discard identical copies if they reference the same server object. This gains even more significance as modified proxies may also contain cached data. If no precautions are taken clients proverbly drown in cached attributes. This problem has been realized by others before, e.g. [19], but proposed solutions have not proved satisfactory due to their lack of transparency.

3.2 The Idea

We suggest using a "Client-Side Container", i.e. a static look-up table in each client process that is queried every time a new proxy has to be transferred, although more elaborate caching services like [2] could be used as well for implementation in later versions. The concept is based on *Stub Annotation* as described in Sect.2.2.3 for simplicity because it enables easier short-term integration than a full-scale proxy generation solution. In other words, instead of modifying business interfaces and component implementations, the generated default proxies are extended by self-defined subclasses.

These modified, tool-generated proxies provide adapted caching functionality for the components' attributes and check remote reference return values against the Client-Side Container (see Fig.1). If a remote reference turns out to equal another proxy for the same component that was already stored in the Client-Side Container's repository it will be discarded and the equivalent reference from the repository will be returned. On the other hand, the Client-Side Container transparently stores the new remote reference, if no equal proxy can be found.

3.3 Object Equality in Component-based Middleware Platforms

As mentioned introductorily, special attention must be given to object identity. One might assume that stubs can simply be maintained in a `Hashtable` but there are certain obstacles that component models like EJB additionally introduce because specifications [13,7] imply that the results of `hashCode()` and `equals()` are undefined. Unfortunately, these methods are crucial for proper storage in standard Java hash tables.

`javax.ejb.EJBObjects` are derived from `java.rmi.Remote` which makes them accessible across network boundaries but the very concept of EJB relies on dynamic redistribution of subsequent remote calls for a certain business

object to different servants for pooling purposes. Hence `hashCode()` and `equals()` behave as expected for one and the same `javax.ejb.EJBObject` servant but this could potentially be used for different entity identities by the container's pooling algorithms.

EntityBeans' primary keys are not practical for instance identification because they are only unique for a certain bean type in the context of a single deployment. The way out is to delegate the proxies' `hashCode()` and `equals()` implementations to `javax.ejb.Handle`, a long-lived identity object which can be obtained from every `javax.ejb.EJBObject` as unique reference. Handles are cached as well in a lazy-evaluating way to avoid additional remote calls every time equality is tested.

Utilizing these prerequisites, a first simple Client-Side Container prototype based on a single static hash table was implemented, but possibilities for more elaborate versions are virtually unlimited.

3.4 Usage

To visualize the usage of Client-Side Containers, imagine the following example: An `OrderBean` references a `CustomerBean`. A `getCustomer()` method would be implemented to enable navigating access in an object-oriented manner, returning a remote reference to a `CustomerBean`. The generator tool would realize this dependency, thus inserting code into the generated proxy to perform the above mentioned look-up in the following way:

```
public class _Order_Stub extends __Order_Stub {
  private static transient CSContainer
    cscontainer = CSContainer.getCSContainer();
  private Customer customerCache = null;
  // (...)
  public Customer getCustomer()
  throws java.rmi.RemoteException {
    if (customerCache==null) {
      Customer c = super.getCustomer();
      customerCache = cscontainer.getStub(c);
    }
    return customerCache;
  }
}
```

The `rmic`-generated default stub is renamed to `__Order_Stub` and the augmented implementation that takes its place is derived from that class. Stub instantiation in Java is based on naming conventions which makes renaming necessary. On the other hand, default stubs should not be altered beyond simple renaming because their structure may change without notice due to Sun's internal modifications.

This example also demonstrates the integration of caching functionality. An Order's Customer is normally determined at creation time which makes this attribute a perfect caching candidate.

Things get more complicated if multivalued relationships between entities occur. Due to dynamic class loading, proxy classes are transfered correctly but

the instances have to be checked as well for duplicates with the local client-side container. We ensured this behavior by implementing generic wrappers for `java.util.Collection` and `java.util.Iterator`.

As long as all proxies of a given deployment are modified by the generator tool this concept works well. If standard stub implementations are mixed in they will pollute the positive effects and reinvigorate proliferation of proxies.

Note that no modifications are necessary on existing client or server code because all changes are entirely transparent to the component itself. Not even sources are needed for supplementary integration of caching. Stubs can be regenerated by `rmic` or a container's corresponding tool based on existing class files. These are in turn altered by the stub generator tool. This two step process will be replaced by a single stub generator tool in a later version.

3.5 Deployment

During EJB deployment, two archives, so-called JAR files, are packaged by the container's deployment tool – one for Bean clients containing remote stubs and interfaces, and another one for the server which additionally comprises the bean implementation itself and all necessary skeleton or tie classes.

The question arises how to get modified proxies into EJB archives. Unfortunately, most EJB servers don't even expose generated server JARs during Bean deployment. As long as there are no container-supported access possibilities, client and server JARs have to be unpacked, modified and copied back to their container-specific locations. Some containers like JBoss [4] leverage special optimized, RMI-incompatible protocols that don't even *have* physical proxy classes. They are dynamicly generated using *invocation handlers* of the *Reflection*-API which in turn provides new access points for integration.

So presently, deployment procedures have to be adapted to individual container implementations. We are experimenting with possibilities for tighter integration into the deployment processes of open-source containers like JBoss. Furthermore, we hope that container providers will realize the benefits of transparently caching stubs and start implementing Smart-Proxy-like interfaces or integrating similar caching solutions themselves.

3.6 Consistency Issues

It has been outlined that changes made to replicated distributed objects always result in inconsistencies. Most applications can fortunately cope with these inconsistencies for a short lag. Many publications [1,10,6] elaborated on different consistency levels and protocols. We currently support only dirty reads, i.e. read accesses without obtaining locks on the server. An optimistic locking strategy using time stamps is under development. Our prototype features optional cache invalidation via regular cache purges but we are working

[4] `http://www.jboss.org/`

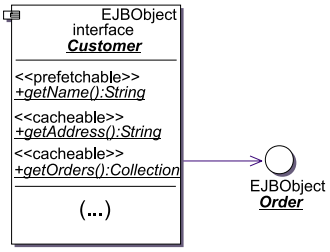

Figure 2. Usage example for UML stereotypes

on an update propagation solution that asynchronously notifies distributed caches of an entity upon write accesses. Future versions will allow descriptive specification of transactional requirements.

4 UML Support

It has already been outlined that most caching concepts lack integration with visual design and builder tools. Caching properties have to be stored externally in proprietary formats. Nevertheless, the *Unified Modelling Language* (UML) provides the means of marking up a component's attributes for different caching strategies. So-called *Stereotypes* (see Fig.2) are put at the designer's disposal. They imply certain characteristics and roles that can be evaluated by code generators and other tools to deduce applicable algorithms and code segments.

The following distinctions between component attributes cover most use cases in respect to caching:

prefetchable these attributes are most frequently needed by the interface's clients, so they should be transferred already at stub / proxy initialization;

group-prefetchable denotes a group of attributes that are to be transferred all together as soon as one of them is queried;

cacheable additional attributes to be cached separately upon first access;

volatile non-cacheable attributes that are subject to frequent changes or that should only be accessed in a transactional context.

The way stereotypes are stored in an UML model completely depends on the design tool's implementation. Design tools can be distinguished in two categories based on their model storage: (1) Special repository formats can be used to store a UML model in an efficient way, e.g. Rational Rose. (2) Source code repositories rely on storing all model elements in corresponding source code segments, e.g. Together. The latter alternative usually lacks efficiency for larger projects but knowing that it uses JavaDoc-like tagged code comments for storing certain model elements provides a possible access point for stub generators. Tools of the other persuasion normally provide exporter plug-ins for generating editable source code from their internal model repository. If these exporters are modified to convert stereotypes into the same code

segments, tools of both categories may be used interchangeably.

5 Future Prospects

First proof-of-concept tests showed the tremendous potential of cached entity attribute access: Queries against a local cached copy took only a few μs whereas equivalent remote call round-trips range one magnitude higher at several ms. Especially communication-intensive JSP / Servlet engines as well as GUI-front-ends benefit from these performance gains.

Our current efforts focus on more differentiate consistency levels and more elaborate cache memory management. Emerging specification proposals like [2] are evaluated in this context.

The main goal of our work is to develop a framework that covers a component's whole life cycle – from design to runtime – with respect to caching issues.

References

[1] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Practice and Experience*, 4(5):337–355, August 1992.

[2] Jerry Bortvedt. *JCache - Java Temporary Caching API.* Oracle Corporation, 19 March 2001. Java Specification Request #107.

[3] Kyle Brown. Session bean wraps entity beans. In *Portland Pattern Repository.* February 2001.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[5] Steve J. Caughey, Graham D. Parrington, and Santosh K. Shrivastava. SHADOWS - a flexible support system for objects in distributed systems. In *3rd International Workshop on Object Orientation and Operating Systems (IWOODS'93)*, pages 73–82, Asheville, NC (USA), 1993.

[6] Gregory Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a caching service for distributed CORBA objects. In *Middleware'00*, pages 1–23, 2000.

[7] Linda G. DeMichiel, L. Ümit Yalčinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification Version 2.0.* Sun Microsystems, 14 August 2001. Final Release.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[9] Daniel Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for Internet cooperative applications. In *Middleware'98*, The Lake District, England, 1998.

[10] Rammohan Kordale, Mustaque Ahamad, and Murthy V. Devarakonda. Object caching in a CORBA compliant system. *Computing Systems*, 9(4):377–404, 1996.

[11] Tony Loton. The smart approach to distributed performance monitoring with Java. *JavaWorld*, 2000.

[12] Paul Martin, Victor Callaghan, and Adrian Clark. High performance distributed objects using caching proxies for large scale applications. In *International Symposium on Distributed Objects and Applications*, pages 110–119, 1999.

[13] Vlada Matena and Mark Hapner. *Enterprise JavaBeans Specification Version 1.1.* Sun Microsystems, 24 November 1999. Final Release.

[14] Sun Microsystems. Design patterns catalog. In *J2EE Design Patterns.* 2001.

[15] Martijn Res. Reduce EJB network traffic with astral clones. *JavaWorld*, January 2001.

[16] Mark Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *6th International Conference on Distributed Computer Systems*, May 1986.

[17] Rob Stevenson and Leanoard Theivendra. *Developing EJB Access Beans in VisualAge for Java.* IBM Toronto Lab, October 2000.

[18] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt, and Ossama Othman. The design and performance of meta-programming mechanisms for object request broker middleware. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[19] M. Jeff Wilson. Get smart with proxies and RMI. *JavaWorld*, November 2000.

[20] Matthew J. Zelesko and David R. Cheriton. Specializing object-oriented RPC for functionality and performance. In *16th IEEE International Conference on Distributed Computing Systems.* Computer Science Department, Stanford University, IEEE Computer Society Press, May 1996.